CREATING POLYNOMIAL DIVISION LOGICAL DEVICES

Field of the Invention

The field of the invention relates to creating logical devices for

5 performing polynomial division. In particular, it relates to a method of using a hardware description language to build code which can be directly synthesized to create a logical device for performing polynomial division and optionally optimizing code using pipelined stages to achieve a maximum operating frequency in the logical device. The polynomial division has applications in

10 calculating a Cyclic Redundancy Check for an m-bit data word, in error correction schemes and in scramblers.

Background to the Invention

A cyclic redundancy check (CRC) is used to detect errors in stored or

15 transmitted data. A CRC is calculated using an algorithm which divides the data block by a generator polynomial to produce a remainder. This remainder is the CRC and it is usually appended to the data block before it is transmitted or stored. When the block of data with the appended CRC is received or retrieved from storage, the CRC algorithm is applied to the data block for a second time

20 and may be compared with the originally calculated CRC. If they are identical, the data block is said to contain no errors with a known confidence interval. If the two CRCs do not match, then the data contains an error.

CRC algorithms and their methods of application are known. In one method, known as the feedback method, the message or block of data for

25 which the CRC is calculated must be augmented by appending n zero's to the message, where n is the number of bits in the CRC (i.e. the order of the generator polynomial). An alternative for implementing a CRC algorithm known as the feed forward method does not require augmentation of the message.

A CRC calculation can be implemented in hardware using shift register

30 arrangements which are known. In the feedback method, n zeros are pushed through the register after the original message. Using the feed forward method, the CRC registers contain the CRC value for the message as soon as the last bit of the message arrives. This is particularly useful in transmission systems

where the CRC must be appended immediately after the message or data block. In contrast, using the feedback method, the system would need to wait an additional n clock cycles until the CRC calculation is complete.

CRC circuits can be built using digital logic and may be implemented in programmable logic devices such as Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs). This hardware can be designed either with schematics or using a language which describes digital electronic systems such as VHDL (VHSIC Hardware Description Language) and Verilog®. Once the VHDL or Verilog® code is written to describe the desired behavior of the digital logic circuit, it can be synthesized by using a development tool suitable for the target device to create the final hardware with the desired function or behavior. Digital logic CRC calculators can also be designed to work on a word or an m-bit bus instead of a single bit serial stream. Such CRC calculators are often created using tools such as code generators. For example, to design a CRC calculator capable of working on an m-bit bus, a code generator, for example written in 'C', Perl or other programming language is used. Code generators generate synthesizable High Level Description language (HDL) code such as VHDL or Verilog® based on specific input parameters which are accepted by the code generator. In the case of a CRC calculator for an m-bit bus, these parameters are (i) the CRC generator polynomial and (ii) data bus width. Therefore, the code generator generates HDL code for a logic circuit specific to the CRC generator polynomial and word or data bus width that are passed as inputs to the code generator program.

A limitation of this method of creating CRC calculators is each time a new CRC generator polynomial (i.e. a polynomial with more or less terms) is to be used in the CRC calculator, a designer must use the code generator to produce new HDL code suitable for that polynomial and data word width combination. That is, the HDL code created by the code generator is specific to each CRC polynomial and data word width combination. Therefore the HDL code is not transportable across a range of combinations and unique HDL code is required for every polynomial and data word width combination. This consumes valuable programming resources and ultimately results in an

undesirable delay between identifying a new generator polynomial and word/bus width combination and providing a suitable CRC calculator circuit.

In addition to these limitations, the code produced using existing methods for creating a CRC calculator results in logic with maximum operating
5 frequencies which are less than desirable for real time calculation of CRCs in certain applications. Further, they do not allow the designer to make a trade off between the maximum frequency of operation and silicon area, using a single value setting that is specified as a parameter that is passed to the HDL code module.
10

Summary of the Invention

A method according to a first embodiment of the present invention creates a logical device performing polynomial division. The method includes using a hardware description language to build code directly describing
15 synthesizable logic for performing the polynomial division and implementing the logic on a target device. The code receives as inputs a parameter identifying a polynomial and a parameter identifying a number of data bits for which the polynomial division is performed.

A method according to a second embodiment of the invention creates a
20 logical device performing polynomial division for a given n-degree polynomial. The polynomial division includes calculating a next n-term remainder for an incoming data unit having d terms. The method includes creating code using a high level description language directly describing synthesizable logic for performing the polynomial division. The polynomial division includes
25 automatically extracting a subset of data terms for calculating each of the next remainder terms. The next remainder is then calculated by performing a logical XOR operation on the subset of data terms with a subset of relevant remainder terms calculated for a preceding data unit. The method further includes implementing the logic on a target device. Preferably, the XOR operation
30 includes a pipelined XOR operation with a pre-definable number of pipeline stages, the pipelined XOR operation operating on the subset of data terms. Pipelining may improve performance of the logical device.

4

According to a third embodiment of the invention, a computer program product residing on a programmable medium contains hardware description language code directly describing synthesizable logic suitable for implementation on a target device. The target device conveys a programmed

5   method of performing polynomial division on units of data each having d terms and using an n-degree polynomial to calculate a next remainder having n terms. The programmed method includes automatically extracting a subset of data terms for calculating each of the next remainder terms and calculating the next remainder by performing a logical XOR operation on the subset of data

10   terms with a subset of relevant remainder terms calculated for a preceding data unit.

Brief Description of the Drawings

Figure 1 is a block diagram showing a prior art process of creating a

15   logical device for calculating a CRC.

Figure 2 is a block diagram showing steps used in creating a device for calculating a CRC according to an embodiment of the present invention.

Figure 3 shows an equivalent logic circuit which results after synthesis of the `calc_crc()` function according to an embodiment of the invention.

20   Figure 4 illustrates an equivalent logic circuit which results after synthesis of the `calc_crc()` function for an embodiment of the invention using a polynomial P=10101, for a 4 bit CRC and a databus 8 bits wide.

Figure 5 is a block diagram showing a prior art process of manually incorporating pipelined stages into the HDL code.

25   Figure 6 illustrates an equivalent logic circuit for an embodiment of the invention incorporating the functions `identify_data_terms()` and `identify_crc_terms()` which select the data bits and crc bits required to calculate each of the new crc bits automatically.

Figure 7 illustrates the equivalent logic circuit of Figure 6, further

30   including a pipelined XOR gate having a number of pipeline stages determined by the parameter "number_of_pipeline_stages".

Detailed Description

     Figure 1 illustrates steps involved in creating a CRC calculating device using the prior art methodology. Firstly, a designer is provided with specifications for a desired CRC calculating device, namely the polynomial P

5   and bus width B. Given these constraints, the designer uses a code generator to generate hardware description language (HDL) code which describes the functionality of the CRC calculating device for that specific polynomial and data-bus width combination P and B. Code generators use a range of standard programming languages such as 'C', Awk, Perl and Java.

10   The HDL code generated by the code generator is then used by a tool such as a "synthesizer" and "place and route tool" to create a configuration file for use by a machine which implements the logic procedure on the programmable target device. Accordingly, the configuration file is usually not readable by humans.

15   Using the prior art method, designing a CRC calculating device was at least a two step process. First, designers used the code generator, say in 'C', to generate HDL code for particular polynomial and data-bus width combination. Second, they would use the HDL code as input to a synthesis tool to create a configuration file for synthesizing the programmable logic to be

20   implemented on the target device. This slowed the development cycle whenever a new polynomial/bus width combination was required. Designers would usually retain a copy of each version of the code generator program and the resulting HDL code to track the development cycle which also consumed resources.

25   Referring now to Figure 2, according to an embodiment of the invention, a method of creating a logical device performing polynomial division includes using a hardware description language to build code 2 directly describing synthesizable logic as exemplified in Figure 4 for performing the polynomial division. The method includes implementing the logic on target device 5 using

30   synthesizer/place and route tool 3 which creates non-human readable configuration file 4. Using the method, code 2 receives as inputs parameter P identifying a polynomial and parameter B identifying a number of data bits for which the polynomial division is performed.

The hardware description language may be any suitable language. The hardware description language used in the description of this invention is VHDL although other hardware description languages such as Verilog® may be used as an alternative. The HDL is used to describe synthesizable logic for

5   implementation in programmable logic devices such as Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs) and other suitable target devices. The programmed or manufactured target device can then be used to perform polynomial division.

The invention will now be described with reference to the creation of

10   logical devices for calculating CRCs. However, it is to be understood that the invention is also applicable to the creation of logical devices suitable for other applications including but not limited to scramblers and error correction devices and schemes.

Using the method of the present invention, a CRC calculating device can

15   be designed for calculating CRCs for any given generator polynomial and data-bus width combination. The term "data-bus" may be used interchangeably with "data-word". Therefore, the method of the present invention can also be used to design a CRC calculator for calculating CRCs for any generator polynomial and data-word width combination. The data-word/data-bus may have a width being

20   any number of bits but most commonly, the number of bits that make up the data bus is an integer power of 2 (2^i, where 'i' is an integer) such as 8, 64, 128, 256 etc.

Figure 2 shows the steps involved in creating a CRC calculating device according to an embodiment of the invention. Again, the designer is provided

25   with the polynomial P and data-bus width B for which the CRC calculating device is to be created. However, the designer does not use a code generator to generate HDL code built around P and B. Rather, P and B are used directly as inputs to generic HDL code which has been created to describe the behavior or structure of the CRC calculating device, regardless of the polynomial and

30   data bus-width combination.

In one embodiment of the invention, a code fragment identified as `calc_crc()` in Appendix 1 describes a logic procedure for calculating an 'n' bit CRC for a 'B' bit data bus using the polynomial 'P' when given "data_in",

being the next word on the data bus and "crc_in", being the crc that resulted from processing the previous data_in word.

For example, the calc_crc function can be used to generate the required logic to calculate a 4 bit CRC for an 8 bit data bus, using generator polynomial

5    $x^4 + x^2 + x^0$ (which can be represented as vector 10101). Using these input parameters as an example, the HDL code in `calc_crc()` describes the logic procedure for calculating each of the n CRC bits as follows.

```
10   crc_out(3) = data_in(7) xor data_in(5) xor data_in(4) xor data_in(3)
                  xor data_in(1)
                  xor crc_in(3)

     crc_out(2) = data_in(6) xor data_in(4) xor data_in(0)
15                xor crc_in(2) xor crc_in(0)

     crc_out(1) = data_in(7) xor data_in(3) xor data_in(1)
                  xor crc_in(1) xor crc_in(3)

20   crc_out(0) = data_in(6) xor data_in(2) xor data_in(0)
                  xor crc_in(2)
```

In an embodiment of the invention, the `calc_crc()` function is

25   incorporated into the invention's HDL code in such a way that output of the `calc_crc()` function is latched after processing each word. Figure 3 shows the equivalent logic circuit which results after synthesis. Figure 4 illustrates the equivalent logic circuit which results after synthesis for the parameters exemplified where polynomial P=10101, n=4 and B=8.

30   It can be seen that using the method of the present invention, there is no need to separately use a code generator to generate HDL code specific to the polynomial and data-word width combination for which the CRC calculator is being designed. Rather, the values for P and B are input to the code as parameters. Preferably, B is set automatically depending on the number of bits

35   in `data_in`.

The HDL code is then processed by the synthesis/place and route tool 3 to create a configuration file 4 specific to the polynomial P and data word-width B combination. The required logic is then implemented on the target device resulting in programmed target device 5.

It is to be noted that as wider data-bus widths and larger polynomials are used, the complexity of the XOR logic created for each CRC bit increases. As a consequence in the target device, a signal takes longer to propagate through the many interconnected gates on the silicon wafer from which device 5 is

5 constructed and the maximum achievable clock speed of the design decreases. In an alternative embodiment, designers can improve the operation of a polynomial division logical device (such as a CRC calculating circuit) created according to embodiments of the invention by breaking up the XOR logic function and pipelining sections of it.

10 Preferably, in addition to generating configuration file 4, synthesizer/place and route tool 3 also generates a report file 6 providing performance characteristics exhibited by programmed target device 5. Such characteristics include the maximum frequency of operation. This can be used to determine the number of pipelined stages which should be included in the

15 logic to improve the performance of the logic device.

In prior art systems pipelined stages were added into the HDL code manually. This process is outlined by the steps shown in Figure 5. The designer inspected report file 6 to determine if the operational speed of programmed target device 5 would be sufficient for use in a particular system. If

20 the clock speed was insufficient, the designer manually inspected the HDL code and inserted 1 or more pipelined stages into the logic design by inserting additional code fragments in the HDL. This divided the logic into segments, each of which were designed to be performed in a single clock cycle. Although the inclusion of one or more pipelined stages increased the device's

25 operational frequency, the process by which the pipelined stages were incorporated into the design was laborious and time consuming using prior solutions.

In one embodiment of the invention, the number of pipeline stages can be modified simply by changing the value of an input parameter and re-

30 synthesizing the digital logic.

According to a second embodiment of the invention, a method creates a logical device performing polynomial division for a given n-degree polynomial and includes calculating a next n-term remainder for a data unit having d terms.

Code is created using a high level description language directly describing synthesizable logic for performing the polynomial division. The n term remainder is calculated by automatically extracting a subset of data terms (using code fragment as `identify_data_terms()` in Appendix 2) for

5 calculating each of the next remainder terms. The next remainder term is then calculated by performing a logical XOR operation on the subset of data terms with a subset of relevant remainder terms which has been calculated for a preceding data unit. The directly synthesizable logic is then implemented on a target device 5.

10 Preferably, the subset of remainder terms is automatically extracted by, for each of the remainder terms, identifying remainder terms calculated for a preceding data unit which are required to calculate the next remainder, and performing a logical AND on the identified remainder terms with the remainder terms calculated for the preceding data unit. However, any other suitable

15 method for extracting the subset of relevant remainder terms may be used.

As demonstrated in the example provided above, the logic equation required to generate each crc bit is the logical XOR of a subset of the data_in bits with a subset of the current crc bits, crc_in. Therefore, when pipelining stages of the logic device, instead of using the function `calc_crc()` to

20 generate the logic equation for calculating each crc bit, a new function can be used. The new function identifies the bits of data_in that are logically XOR'd together when calculating each crc_out bit.

A code fragment identified as `identify_data_terms()` in Appendix 2 describes a suitable logic procedure to achieve this. The output of this

25 procedure is a set of 'n' vectors of 'B' bits each, where n is the number of bits in the desired CRC. Therefore, there is 1 vector for each crc bit to be calculated. Each bit of a vector represents 1 bit of data_in. If the bit corresponding to a particular data_in bit is '1' then this indicates that the data bit is included in the XOR equation for the related crc_out bit. If the bit corresponding to a particular

30 data_in bit is '0' then this indicates that the data_in bit is not included in the XOR equation for calculating the related crc_out bit.

To demonstrate by way of example, for the crc polynomial and data bus width combination P and B exemplified above, the vectors resulting from the `identify_data_terms()` procedure are as follows.

```
5          data_xor_ena_vec(3)  = 10111010
           data_xor_ena_vec(2)  = 01010001
           data_xor_ena_vec(1)  = 10001010
           data_xor_ena_vec(0)  = 01000101
```

10   Another function is used to identify the bits of crc_in that are logically XOR'd together when calculating each crc_out bit. A code fragment identified as `identify_crc_terms()` in Appendix 3 describes a suitable logic procedure to achieve this. The output of this procedure is a set of 'n' vectors having 'n' bits, n being the number of bits in each crc term. Therefore, there is 1 vector for each crc bit to be calculated. Each bit of a vector represents 1 bit of crc_in. If the bit corresponding to a particular crc_in bit is '1' then this indicates that the crc_in bit is included in the XOR equation for the related crc_out bit. If the bit corresponding to a particular crc_in bit is '0' then this indicates that the crc bit is not included in the XOR equation for calculating the related crc_out bit.

Accordingly, using the same polynomial and data-bus width combination P and B already exemplified the vectors resulting from the `identify_crc_terms()` procedure are as follows.

```
25         crc_xor_ena_vec(3)  = 1000
           crc_xor_ena_vec(2)  = 0101
           crc_xor_ena_vec(1)  = 1010
           crc_xor_ena_vec(0)  = 0100
```

30 The functions `identify_data_terms()` and `identify_crc_terms()` can be incorporated into the HDL code, enabling the data bits and crc bits required to calculate each of the new crc bits to be selected automatically. To calculate the new crc bits, the terms identified by the functions `identify_data_terms()` and `identify_crc_terms()` are logically
35 XOR'd together. The value of the new crc bits is then latched after each word is

processed. A logic circuit illustrating this process for one of the 'n' crc bits is provided in Figure 6.

By performing the process of logically AND-ing each data_in bit with it's corresponding data_xor_ena_vec bit, it can be seen that irrelevant data_in bits are disregarded or effectively masked out from the logical XOR function calculating the nth crc bit (since Y XOR '0' = Y). Other suitable methods for eliminating irrelevant data_in bits from the input to XOR gate may be used as an alternative. After synthesis, the circuit of Figure 6 is reduced to a logic circuit which is identical to that which is generated using the `calc_crc()` function of Appendix 1.

Advantageously, in one embodiment of the invention illustrated in Figure 7, the maximum operating frequency of the XOR logic procedure can be improved by employing a pipelined XOR element 8 (which is known in the art) instead of a simple XOR element 7 (as in Figure 6), to combine the masked data_in bits with the masked crc_out bits. This allows the number of pipeline stages used in the calculation of the crc to be varied simply by changing a parameter. This parameter can be provided directly to the HDL code and is identified in Figure 7 as "number_of_pipeline_stages". A final circuit described by such HDL code before synthesis is illustrated in Figure 7. If the number of pipeline stages is zero, the circuit will operate as the non-pipelined circuit of Figure 6.

While the embodiments of the present invention have been illustrated in detail, it should be apparent that modifications and adaptations to these embodiments may occur to one skilled in the art without departing from the scope of the present invention as set forth in the following claims.